MOSER
PATTERSON &
SHERIDAN, LLP
ATTORNEYS AT LAW

3040 Post Oak Blvd, Suite 1500
Houston, TX 77056-6582
TEL 713.623.4844
FAX 713.623.4846
WWW.MPSLLP.COM

RECEIVED
CENTRAL FAX CENTER
AUG 29 2005

FACSIMILE COVER SHEET

DATE:	August 29, 2005
FILE NO:	ROC920000286US1
TO:	Examiner Qamrun Nahar
FAX NO:	571/273-8300
COMPANY:	USPTO
FROM:	Randol W. Read
PAGE(S) with cover:	25
ORIGINAL TO FOLLOW?	☐ YES ☒ NO

APPEAL BRIEF

U.S. SERIAL NO.:	**09/814,620**
FILING DATE:	**March 22, 2001**
INVENTOR:	**Hanson et al.**
EXAMINER:	**Qamrun Nahar**
GROUP ART UNIT:	**2191**
CONFIRMATION NO.:	**3982**

RECEIVED
OIPE/IAP
AUG 30 2005

368309_1

PATENT
Atty. Dkt. No. ROC920000286US1

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE BEFORE THE BOARD OF PATENT APPEALS AND INTERFERENCES

RECEIVED
CENTRAL FAX CENTER
AUG 29 2005

In re Application of: Hanson et al. §
§ Group Art Unit: 2191
Serial No.: 09/814,620 §
Confirmation No.: 3982 §
§ Examiner: Nahar, Qamrun
Filed: March 22, 2001 §
For: Method, Article of Manufacture and Apparatus for Performing Automatic Intermodule Call Linkage Optimization §

MAIL STOP APPEAL BRIEF-PATENTS
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

CERTIFICATE OF FAX
37 CFR 1.8

I hereby certify that this correspondence is being facsimile transmitted to the Patent and Trademark Office to fax number 571/273-8300 to the attention of Examiner Qamrun Nahar on the date below:

August 29, 2005
Date

Randol W. Read

Dear Sir:

APPEAL BRIEF

Applicants submit this Appeal Brief to the Board of Patent Appeals and Interferences on appeal from the decision of the Examiner of Group Art Unit 2191 dated March 25, 2005, finally rejecting claims 1-29. The final rejection of claims 1-29 is appealed. This Appeal Brief is believed to be timely since facsimile transmitted by the due date of August 27, 2005, as set by mailing a Notice of Appeal on June 27, 2005. Please charge the fee of $500.00 for filing this brief to Deposit Account No. 09-0465.

08/30/2005 JBALINAN 00000020 090465 09814620
01 FC:1402 500.00 DA

RECEIVED
CENTRAL FAX CENTER
AUG 29 2005

TABLE OF CONTENTS

Real Party in Interest

The present application has been assigned to International Business Machines Corporation, Armonk, New York.

Related Appeals and Interferences

Applicants assert that no other appeals or interferences are known to the Applicants, the Applicants' legal representative, or assignee which will directly affect or be directly affected by or have a bearing on the Board's decision in the pending appeal.

Status of Claims

Claims 1-29 are pending in the application. Claims 1-28 were originally presented in the application. Claims 1-29 stand finally rejected as discussed below. The final rejections of claims 1-29 are appealed. The pending claims are shown in the attached Claims Appendix.

Status of Amendments

The Examiner declined to enter amendments submitted after final.

Summary of Claimed Subject Matter

Claimed embodiments of the invention provide for methods, apparatus and articles of manufacture configured to optimize a computer program. *Application*, 6:24-28. Embodiments of the invention optimize a computer program by selecting a call linkage that is the most efficient for each procedure call in the source code, including intermodule call linkage. *Id.*, 6:20-29; 3:14-17.

Regardless of the data-types provided by a particular programming language (e.g., integers, characters, real numbers, or pointers), parameter arguments specified in a procedure call must be made available to the object code generated by the compiler for the procedure call. A call linkage describes the mechanism used to provide parameter arguments to the procedure call. *Id.*, 6:30-31.

Claimed embodiments optimize a computer program during compilation by selecting the most efficient call linkage for procedure calls. *Id.*, 6:20-29. During compilation, claimed embodiments include extracting information for each procedure call contained in the source code. *Id.*, 11:29-35 – 12:1-3; *see also*, Figures 4A-4B; Figure 5, method step 504 and 506, creating data structures to store extracted information (further illustrated in Figures 6 and 7, respectively). By analyzing the information extracted from the source code, a call linkage is selected for each procedure. *Id.*, 12:4-10, Figure 5, method-step 508 (further illustrated in Figure 8). The compiler generates object code according to the selected call linkage. *Id.*, 12:9-10. When the program is executed, the program runs according to the selected call linkage. *Id.*, 12:9-10.

Claimed embodiments include selecting from at least a register-based call linkage and a memory-based call linkage. *Id.*, 6:30-34 – 7:1-9. Figure 9A (object code for a memory-based call linkage), Figure 9B (object code for a register-based call linkage). While a compiled program is executing, a memory-based call linkage passes parameters using main memory. *Id.*, 6:30-34 – 7:1-4. To pass arguments using a memory based call linkage, the arguments are first moved from registers to memory; program control then branches to the procedure, where the arguments may be moved from memory back into registers. Thereafter, the parameter values are used by the

instructions of the procedure call. *Id.*, 7:1-4. In register-based call linkage, procedure arguments are copied to parameter registers, control then branches to the procedure, where the arguments may be moved from parameter registers back into program registers. *Id.*, 7:4-9.

Because registers operate much more quickly than memory, register-based call linkage is typically preferred, but not always possible or desirable. *Id., 10:23*-29. Because of this, embodiments of the invention optimize object code by selecting the most efficient call linkage for a particular procedure. In addition to memory-based and register-based call linkage, other types of call linkages are possible. *Id.*, 7:16-27.

Issues

Whether claims 1-29 are unpatentable under 35 U.S.C. § 102(e) over *Ungar* (U.S. Patent No. 6,085,035).

Arguments

Anticipation of Claims 1-29 over *Ungar*

Statement of Applicable Law

A rejection under 35 U.S.C. § 102(e) requires that each element be disclosed by the cited reference. Specifically, the Federal Circuit has held that "[a] claim is anticipated only if each and every element as set forth in the claim is found, either expressly or inherently described, in a single prior art reference." *Verdegaal Bros. v. Union Oil Co. of California*, 814 F.2d 628, 631, 2 USPQ2d 1051, 1053 (Fed. Cir. 1987). "The identical invention must be shown in as complete detail as is contained in the ... claim." *Richardson v. Suzuki Motor Co.*, 868 F.2d 1226, 1236, 9 USPQ2d 1913, 1920 (Fed. Cir. 1989). The elements must be arranged as required by the claim. *In re Bond*, 910 F.2d 831, 15 USPQ2d 1566 (Fed. Cir. 1990). Respectfully, the Examiner's rejection fails to satisfy the statutory requirement of 35 U.S.C. § 102(e).

The Reference (Ungar)

Ungar discloses a method for optimizing the object code generated by a compiler, using the data types associated with variables (e.g., character, pointer, integer, and real). In particular, *Ungar* teaches a method for optimizing object code based on variable types, and variable usage patterns. *Ungar*, 3:30-35. In programming languages, a variable is a place to store a value. When a program declares a variable, the compiler typically sets aside a space in memory to store the value given to that variable.

In a statically-typed language, the storage space is dependent on the data type of the declared variable. Some programming languages allow for the data type associated with a variable to change over the course of program execution (commonly referred to as "weakly-typed" or "externally-tagged." That is, the same variable may be used to store different data types at different points during the execution of a computer program. Additionally, a pointer variable may be used as a reference to different data types.

Ungar discloses a method for optimizing the object code generated by a compiler using the data types associated with variables. In particular, the optimization techniques of *Ungar* describes variable types as "mutable" and "immutable." As used by *Ungar*, a variable may be "immutable" or "mutable" depending on whether a variable can (or actually does) store different data types. *Ungar*, 8:11-52, Figure 3. Using a dynamic compilation environment (i.e., the compiler runs concurrent with program execution) the method taught by *Ungar* monitors the executing program to determine actual, or likely variable types and usage patterns, and creates compiled object code accordingly. *Ungar*, 8:35-40. *Ungar* also teaches that in a non-dynamic environment "using a static compiler, the program is first profiled and the optimization process is invoked during a subsequent compilation that utilizes the profiled data." *Ibid.*

Fundamentally, *Ungar* discloses an optimization technique that includes determining the type usage pattern for a variable accessed by a procedure, where the variable may take on different data-types during execution. Once a "preferred" type is determined, the compiler may provide an optimized routine for the preferred type. Further, the compiler may be configured to generate multiple versions of the routine,

based on the different data types that may be assigned to a variable. In describing this optimization technique, *Ungar* defines a number of relevant terms.

- Access–Access to a variable includes both a read access and a write access or any combination of read/write access to retrieve a data-value from, or store a data-value into the variable. *Ungar*, 4:40-45.
- Called routine--A called routine is an OOP method (or similar procedure) that is invoked from a call site, performs an operation and returns to the call site. The called routine may receive arguments passed by the call site. It may also return values. 4:55-58
- Call site--A call site is the procedure used to invoke a called routine. *Ungar*, 4:59-60.
- Data type–The type of the data-value. The data type is either associated with the data-value itself, or with the variable that contains the data-value. There are primary types (for example integer and real) and constructed types (such as those defined by data structures). Some variables store data-values of only one type. These variables are immutable type variables (all staticly typed variables are immutable type variables). Mutable type variables store data-values of different types. Data-values that are of an untagged primitive type have no intrinsic type information available to the executing program. Data-values that are of a tagged primitive type include intrinsic type information within the memory field used to store the data-value itself. Externally tagged primitive types store the type information separately from the data-value itself. Information from both tagged and externally tagged data types are available to the executing program. *Ungar*, 5:1-18.
- Data-value--The data-value is a pattern of bits that have a meaning that depends on the data type associated with the data-value. *Ungar*, 5:19-21.

By monitoring the actual usage of a pointer variable, or an "externally tagged primitive type," patterns may emerge that establish a probability that a given variable is used to store a particular data type. By identifying these patterns, the compiler may be able to generate optimized object code.

Applicants Arguments (Claims 1, 9, 14, and 29)

Claims 1, 9, 14, and 29 each include an element for optimizing object code that includes selecting a call linkage between a caller procedure and a callee procedure. In other words, each of these claims includes selecting the call linkage mechanism that

should be used to pass parameter arguments to the procedure call. As described above, a call linkage describes the mechanism used to provide parameter arguments to the procedure call.

In formulating a rejection with reference to this element recited by claims 1, 9, 14, and 29, but without providing any analysis thereof, the Examiner cites the following passage from *Ungar*:

> A first preferred embodiment optimizes both a called routine and the call site dependent on the types of the data-value passed from the call site to the called routine. Because data-values contained in passed entities (that is, the data-values contained in variables and/or the addresses of the variables themselves) can be specified as arguments to, or a result from, a called routine, the call site generally contains code to select which executable version of the called routine to invoke dependent on the types of the passed entities. The invention detects variables that have immutable types (from the `determine type usage pattern` procedure 305) and optimizes both the called routine and the call site dependent upon the type-mutability of the passed entities. Additionally, if the variables have mutable types, the invention generates multiple versions of the called routine (each optimized for a preferred type as determined by the `determine type usage pattern` procedure 305) that are invoked dependent on the types of the passed data-values. Often, one of these called routine versions is not optimized with respect to any of the passed data-values and so is capable of processing any pattern of types of the passed data-value..

See Final Rejection, p. 4-5, (*quoting Ungar*, 8:52-67 to 9:1-7). Respectfully, this material fails to disclose selecting a call linkage. In fact, the cited passage fails to describe any aspects of call linkage. Rather, it describes optimizing "both a called routine and the call site dependent on the types of the data-value passed from the call site to the called routine." Using the express definitions provided by *Ungar*, this material discloses optimizing a "the procedure used to invoke a called routine," and also optimizing the called routine (i.e., "an OOP method (or similar procedure) that is invoked from a call site."). Specifically, *Ungar* teaches generating multiple versions of the called routine, each optimized for a different data type that may be passed to the called routine. In addition, the procedure invoking the called routine is optimized to invoke one of the different versions of the called routine, based on the data types actually passed at runtime. The optimization is dependent on the types of the data-value passed from the procedure used to invoke a called routine.

Because some programming languages allow for externally tagged types, until the procedure is actually called at runtime, it may be unknown what type of variable is available. Accordingly, "the call site generally contains code to select which executable version of the called routine to invoke dependent on the types of the passed entities." Thus, one called routine may be used to generate multiple executable versions of the routine. During runtime, different versions of the routine are invoked based on the types of parameters actually present in a particular procedure call.

Optimizing a procedure by generating multiple versions, each for different data types, is directed to optimizations flowing from the structure of the source code, i.e., on what variables (and their types) are passed to a procedure call or method invocation. None of this material is directed to selecting a call linkage, i.e., to a mechanism for actually performing the procedure call at the level of the CPU, registers and memory, by moving data values between memory and registers.

In short, selecting a call linkage determines the mechanism used to pass parameters to a procedure: memory-based call linkage stores the values in memory; register-based call linkage stores this data in parameter registers. Nothing in the material cited by the Examiner discloses selecting a call linkage, or call linkage at all. Rather, it discloses compiling multiple versions of a called procedure with different versions for variables that may be used to store different data-types (e.g., the "mutable type," "immutable type," or "preferred type). *Ungar* does not teach or suggest any optimizations regarding the mechanism used to actually pass parameters to a procedure call, (i.e., it is silent on selecting a call linkage). Therefore, because the claimed elements are not taught by the cited passage, allowance of claims 1, 14, 21 and 29 is respectfully requested.

The Applicants presented these distinctions to the Examiner. *See Response to Office Action, dated August 23, 2004, p. 13-15.* In an ensuing Final Office Action, the Examiner responded by stating:

> Examiner strongly disagrees with applicant's assertion that Ungar fails to disclose the claimed limitations recited in claims 1-13. Ungar clearly shows each and every limitation in claims 1-13.

> Ungar teaches extracting information for each procedure call contained in the source code (column 8, lines 11-41). ...

See Final Office Action, p. 14. In fact, Applicants' agree that *Ungar* discloses extracting information for procedure calls in source code. Specifically, *Ungar* discloses obtaining type-usage patterns of a variable. However, this misses the point: *Ungar* fails to disclose selecting a call linkage. Respectfully, the Examiner fails to respond to Applicants' arguments regarding this limitation.

Appellants Arguments (Claims 2, 16, and 22)

As demonstrated by the above discussion, *Ungar* fails to disclose selecting a call linkage. Claims 2, *16, and* 22, further specify that the selected call linkage is one of a memory-based call linkage and a register based call linkage. To clarify the term "call-linkage," Applicants amended Claims 1, 14, and 21, in a Response to Final Office Action mailed on March 25, 2005. As amended, these claims include the limitation originally specified by claims 2, 16, and 22 (and a similar limitation was added to claim 29).

Similar to the rejection of claims 1, 14, and 21, the Examiner simply cites a wide swath of *Ungar* without providing any analysis as to how *Ungar* discloses the claimed limitation: "selecting one of a memory-based call linkage and a register based call linkage". In fact however, the cited passage (specifically, *Ungar*, 9:22-67 – 10:1-21) describes Figure 4B, which "illustrates an optimization process ... for a dynamic compiler utilizing the invention [of *Ungar*]". *Ungar* 9:22- 24. This material describes selecting an "optimization procedure" that "determines whether the source code will invoke a called routine to access a value, or if the routine accesses the variable directly." *Id*. The routine may then be compiled depending on the preferred data types of the variable. Thus, the cited passage describes an optimization that may be applied based on how procedure call accesses a particular variable, e.g., directly or as part of a call to another procedure, and based on the data type of variable being passed. How the compiler selects a mechanism used to pass data values to a procedure call, i.e., selecting a memory-based or register-based call linkage, is simply not addressed.

Therefore, because the claimed elements are not taught by the cited passage, allowance of claims 2, 16, and 22 is respectfully requested.

Applicants Arguments (Claims 3 and 23)

Claims 3 and 23 further characterize executing a procedure call compiled using a memory based call linkage. Specifically, these claims recite the following steps:

allocating a block in a memory to store a value for each argument in the particular procedure call;

storing the value for each argument from a register in a processor to the block in the memory;

branching the procedure call to a callee procedure; and

loading the value for each argument from the block in the memory back to the register.

Again without any analysis, the Examiner cites a passage from *Ungar*. Specifically, the Examiner cites the following:

> If the accessed variables are type mutable, the `optimize source for called routine access` procedure 443 determines the preferred data types passed between the called routine and the call site. It then generates a number of versions of called routines each optimized to process data-values having the preferred types. Next, the call site is modified to determine the type pattern of the arguments passed to the called routine and to invoke the corresponding optimized called routine. A general (not optimized dependent on the type pattern of the passed arguments) version of the called routine will be invoked if none of the optimized called routines match the type pattern of the passed arguments. Once the `optimize source for called routine access` procedure 443 completes, the optimization process 430 completes through the `end` terminal 437.

Ungar 10:7-21.

This material is clearly directed to compiling a "number of versions of called routines" that are, "each optimized to process data-values having the preferred types." Once multiple versions of the called procedure are available, the call site (i.e., the procedure invoking the procedure call) "is modified to determine the type pattern of the arguments passed to the called routine and to invoke the corresponding optimized called routine." Nothing in this passage, however discloses performing a memory

based call linkage in a running program or anything about call linkage in the multiple versions of a called routine. The recited steps of allocating a block in a memory ... storing the value for each argument from a register in a processor ... , branching the procedure call to a callee procedure; and loading the value for each argument back to the register, are simply not disclosed by this cited passage. Therefore, because the claimed elements are not taught by the cited passage, allowance of claims 3 and 23 is respectfully requested.

Applicants Arguments (Claims 4 and 24)

Claims 4 and 24 further characterize executing a procedure call compiled using a memory based call linkage. Specifically, these claims include the following steps, performed by an executing program:

copying a value, for each argument in a procedure call, from a register in the processor to a parameter register in the processor;

branching the procedure call to a callee procedure; and

copying the value from the parameter register back to the register.

To reject these claims, which include a copying step, a branching step, and a copying step, the Examiner refers to the following 6 lines from Ungar:

> However, if the type is immutable, the 'optimize source for local variable access' procedure 441 compiles the source optimized for only that specific type. Once these operations have been compiled the optimization process 430 completes through the 'end' terminal 437.

Ungar, 9:60-65. The recited material describes compiling a single optimized version of a procedure if the data-types of parameters passed to the procedure are immutable, i.e., if the parameters cannot take on different data types. As is plainly evident, the actions disclosed in this passage do not disclose the limitations of claims 4 and 24. Therefore, because the claimed elements are not taught by the cited passage, allowance of claims 4 and 24 is respectfully requested.

Applicants Arguments (Claims 5-13, 15, 17-20, 25-28)

Claims 5-13 are dependents of claim 1 and, therefore, are also believed to be allowable.

Claims 15, 17-20 are dependents of claim 14 and, therefore, are also believed to be allowable.

Claims 25-28 are dependents of claim 21 and, therefore, are also believed to be allowable. Allowance of the same is respectfully requested.

CONCLUSION

Ungar does not teach an optimization mechanism for selecting a call linkage (i.e. a memory-based or register-based call linkage) when generating the object code of a computer program. Nor does *Ungar* disclose executing the compiled program, according to the selected call linkage. Accordingly, Appellants respectfully request that the rejections be withdrawn and that the claims be allowed.

Respectfully submitted,

Randol W. Read
Registration No. 43,876
Moser, Patterson & Sheridan, L.L.P.
3040 Post Oak Blvd. Suite 1500
Houston, TX 77056
Telephone: (713) 623-4844
Facsimile: (713) 623-4846
Attorney for Appellant(s)

CLAIMS APPENDIX

1. (Previously Presented) A method for optimizing a run time for an object code generated from a source code, the method comprising:

extracting information for each procedure call contained in the source code;

selecting a call linkage between a caller procedure and a callee procedure for each procedure call using the extracted information, where the selected call linkage is optimized to minimize the run time of the object code generated from the source code;

generating the object code from the source code; and

running the object code using the selected call linkages for each procedure call.

2. (Previously Presented) The method of claim 1 wherein the selected call linkage is one of a memory-based call linkage and a register-based call linkage.

3. (Original) The method of claim 2 wherein, if the memory-based call linkage is selected for a particular procedure call, the running comprises:

allocating a block in a memory to store a value for each argument in the particular procedure call;

storing the value for each argument from a register in a processor to the block in the memory;

branching the procedure call to a callee procedure; and

loading the value for each argument from the block in the memory back to the register.

4. (Original) The method of claim 2 wherein, if the register-based call linkage is selected for a particular procedure call, the running comprises:

copying a value, for each argument in a procedure call, from a register in the processor to a parameter register in the processor;

branching the procedure call to a callee procedure; and

copying the value from the parameter register back to the register.

5. (Previously Presented) The method of claim 2 wherein the selecting comprises:

detecting whether an error exists for the procedure call;

selecting the memory based call linkage if the error is detected for the procedure call; and

selecting the register-based call linkage if no error is detected for the procedure call.

6. (Original) The method of claim 5 wherein the error is detected if the procedure call has a different number of parameters than the callee procedure.

7. (Original) The method of claim 5 wherein the error is detected if a parameter type for a parameter in the caller procedure is different than the parameter type for the parameter at a corresponding position in the callee procedure.

8. (Original) The method of claim 5 wherein the error is detected if a number of arguments in the procedure call is greater than a number of parameter registers used to run the object code.

9. (Original) The method of claim 5 wherein the error is detected if an argument in the procedure call is unpassable in a register.

10. (Original) The method of claim 1 wherein the extracting of procedure call information comprises:

extracting information for each procedure definition contained in the source code.

11. (Original) The method of claim 10 wherein the extracted procedure call information comprises an identifier for a calling procedure and a callee procedure, and the extracted procedure definition information comprises a number of arguments received by the callee procedure and a classification for each argument.

12. (Original) The method of claim 1 wherein the call linkage is selected in a class comprising one of a register stacks call linkage, a system call linkage and a near versus far call linkage.

13. (Original) The method of claim 1 wherein the extracted information is generated in a data structure used to select the call linkage for each procedure call.

14. (Original) An apparatus for optimizing a run time of an object code generated from a source code, the apparatus comprising:

a memory for storing a compiler program; and

a processor comprising a plurality of registers, where a subset of the plurality of registers comprise parameter registers, the processor performing a method upon executing the compiler program in the memory, the method comprising:

extracting information for each procedure call contained in the source code;

selecting a call linkage between a caller procedure and a callee procedure for each procedure call using the extracted information, where the selected call linkage is optimized to minimize a run time of an object code generated from the source code; and

generating the object code from the source code.

15. (Original) The apparatus of claim 14 wherein the object code is executed using the selected call linkage for each procedure call.

16. (Original) The apparatus of claim 14 wherein the determined call linkage is one of a memory-based call linkage and a register-based call linkage.

17. (Original) The apparatus of claim 16 wherein the selecting comprises:

detecting whether an error exists for the procedure call;

selecting the memory based call linkage if the error is detected for the procedure call; and

selecting the registered based call linkage if no error is detected for the procedure call.

18. (Original) The apparatus of claim 14 wherein the extracting of procedure call information comprises:

extracting information for each procedure definition contained in the source code.

19. (Original) The apparatus of claim 18 wherein the extracted procedure call information comprises an identifier for a calling procedure and a callee procedure, and the extracted procedure definition information comprises a number of arguments received by the procedure and a classification for each argument.

20. (Original) The apparatus of claim 14 wherein the extracted information is generated in a data structure used to select the call linkage for each procedure call.

21. (Original) A computer readable medium storing a software program that, when executed by a computer, causes the computer to perform a method comprising:

extracting information for each procedure call contained in a source code; and

selecting a call linkage between a caller procedure and a callee procedure for each procedure call using the extracted information, where the selected call linkage is optimized to minimize a run time of an object code generated from the source code;

generating a object code from the source code; and

running the object code using the selected call linkages for each procedure call.

22. (Original) The computer readable medium of claim 21 wherein the determined call linkage is one of a memory-based call linkage and a register-based call linkage.

23. (Original) The computer readable medium of claim 22 wherein, if the memory-based call linkage is selected for a particular procedure call, the running comprises:

allocating a block in a memory to store a value for each argument in the particular procedure call;

storing the value for each argument from a register in a processor to the block in the memory;

branching the procedure call to a callee procedure; and

loading the value for each argument from the block in the memory back to the register.

24. (Original) The computer readable medium of claim 22 wherein, if the register-based call linkage is selected for a particular procedure call, the running comprises:

copying a value, for each argument in a procedure call, from a register in the processor to a parameter register in the processor;

branching the procedure call to a callee procedure; and

copying the value from the parameter register back to the register.

25. (Original) The computer readable medium of claim 22 wherein the selecting comprises:

detecting whether an error exists for the procedure call;

selecting the memory based call linkage if the error is detected for the procedure call; and

selecting the registered based call linkage if no error is detected for the procedure call.

26. (Original) The computer readable medium of claim 21 wherein the extracting of procedure call information comprises:

extracting information for each procedure definition contained in the source code.

27. (Original) The computer readable medium of claim 26 wherein the extracted procedure call information comprises an identifier for a calling procedure and a callee procedure, and the extracted procedure definition information comprises a number of arguments received by the procedure and a classification for each argument.

28. (Original) The computer readable medium of claim 21 wherein the extracted information is generated in a data structure used to select the call linkage for each procedure call.

29. (Previously Presented) A computer readable medium containing information thereon, comprising:

a compiler program configured to generate an executable program from a plurality of source code modules, wherein the executable program is optimized by:

(i) extracting information for each procedure call contained in the plurality of source code modules, (ii) selecting, using the extracted information, a call linkage between a caller procedure and a callee procedure, wherein the selected call linkage is optimized to reduce a run time of object code modules generated from the source code modules by the compiler program, (iii) linking the object code modules according to the selected call linkage, and (iv) generating the executable program from the object code modules.

RELATED PROCEEDINGS APPENDIX

No copies of decisions rendered by a court or the Board in the related appeal or interference are included as there have been no decisions by the court or the Board.